Aqua Research Senior Design Project
K1 Prototype

Version	Document Owners	Changes	Date
A	Diego Chavez, John Quinlan	Initial document requirements	4/26/20
B	David Kirby	Formatted code with syntax highlighting, spell-checked, updated flow chart	5/4/20

Document Version: B

Introduction

Our team worked closely with Tim Cushman and Rodney Herrington of Aqua Research, LLC. Our team consisted of three engineering students: Diego Chavez, Computer Engineering, John Quinlan, Electrical Engineering, and David Kirby, Electrical Engineering. Our work done here at Aqua Research set out to produce a prototype of a user-interface controller for the new product under development by Aqua Research: The K1 Batch Generator. Our team successfully produced a working prototype. Some issues were faced by the team along the course of development. Confirming the efficiency of the board to meet the requirements was difficult, as the prototyping board did not have one of the primary demux logic chips wired up to power. Addressing this issue demands future efforts, as the board could not be re-wired in the time that we worked on this project.

The K1 Batch Generator is a very important product under development. With this product, deprived countries around the world can have access to purified water through the utilization of automatically produced Chlorine. The design of the controller is paramount to the effectiveness of the final product. Without reliable hardware and software in the K1 Controller, the unit would not be able to automatically monitor production. This feature allows the user to not be technically certified in operating the system. This allows the product to save lives more efficiently and effectively.

Document Purpose

This document contains the system requirements for K1 Chlorine Generator. These requirements have been derived from several sources. It will discuss the K1 description, 2015 requirements that we interpreted, software and hardware that was implemented, and known problems. This document is intended for a future generation to finish our work on the K1 project.

Project Scope

The K1 chlorine generator has been developed by Mr. Herrington as well as designed into the prototype platform for the final stages of hardware design including control elements to be implemented by our team. Aqua Research believes this product is a key component of their product line in making purified water readily available to areas in need of access to clean water in an on-site, easy-to-use and robust manner.

References

Rodney Harrington, “2015 Requirements Document”,
Tim Cushman, “K1 System Schematic”, Doc # K1K2SYS.SCH
Tim Cushman, “K1 K2 Disinfection Controller”, Doc# K1K2.SCH
Tim Cushman, “K1-K2 Controller Wiring Diagram”, Rev. 3/2020
Tim Cushman, “Ox Tank Floats”, Doc# K2FLOAT.SCH
Tim Cushman, “Brine and Oxidant Generator Floats”, K1FLOAT.SCH
Tim Cushman, “K1 K2 Drivers”, K1K2DRV.SCH

Known Problems Faced by 2020 Senior Design Group

1. Digital Multiplexer is not working when pins are driven high or low. We believe this is a problem with the board, but it should not be ruled out this could be a problem with the software. Certain logic chips were not wired to V_{CC}, therefore the demux states couldn't be tested.
2. The Red, Green, and Yellow LEDs could not be turned out without turning off every other digital multiplexer output. This is a design flaw and needs to be fixed to allow the light to be operated independently and showcase the condition of the K1 machine.
3. The GSM module could not be implemented in this due to a hardware design. The pins for the GSM module need to be specifically 10 and 11. All other pins will not work. All GSM code is commented out until this is fixed. Operating this module requires a subscription to the SIM provider, in our case, Speedytalk. A SIM card with some sort of cellular service is required.
4. The microcontroller should be upgraded to an Arduino Uno to allow for more analog and digital ports, as well as increased dynamic memory. Our team ran into a limited supply of dynamic memory while designing our software.
5. Fram needs to be reconfigured to efficiently store the languages. The current implementation wastes too much space.
6. FRAMs need to have one pin addressed to V_{CC} or a high pin to change the address so two FRAMs on I2C can be operated at the same time.

K1 Description

Product Purpose

Water is an essential need for humanity, without water all life on earth would perish. Water can be deceiving and look safe to drink, but may contain things like Cholera, Giardia, Legionella, Shigellosis, among countless others.

The required starting capital to make water safe to drink for third world countries is a significant investment to build a Water Treatment System, and this is frequently not possible for emerging countries. About 790 million people don't have access to safe water around the world (source: *https://www.cdc.gov/healthywater/global/wash_statistics.html*).

Aqua Research offers affordable and sustainable lifesaving technology to improve the quality of living for those who do not have immediate access to clean water. The H2gO Purifier, another device developed by Aqua Research, is a compact and affordable device to clean dirty water. The device has been used to purify water for multiple applications: camping, emergency readiness kits, military applications, hiking, and natural disasters. The recent Flint Michigan Legionnaires disease outbreak is an example of where the H2gO has provided contaminant-free water, saving countless lives.

With the development of an automated chlorine production system, people will no longer need to manage the entire electrolysis production manually. The K1 system will revolutionize the production of chlorine, and therefore save more lives around the world. The design of the K1 Controller is paramount in this development, as efficient software will streamline the production of oxidant in the field.

Design Implementation Constraints

One primary concern is cost. The ideal bill of materials for constructing a K1 system should be less than $1,500. Dynamic memory existent within the microcontroller will be a significant design constraint, as only so much software can be developed before the system ceases to operate properly.

Assumptions and Dependencies

The K1 team assumes the user has enough experience with technology and knows how to connect a stable power source, as well as understand simple troubleshooting procedures. The K1 project needs salt and water to operate properly. Brine must therefore be in steady supply to the user. Another primary assumption is that the user has enough containers to store produced Oxidant as well as temporarily store the required brine.

Requirements

2015 Requirements List

1. Activate system if a low-level oxidant tank switch is activated.
2. Open the brine valve until the electrolyte tank switch is active.
3. Open water solenoid until electrolyte tank brine switch is active.
4. Activate power to the cell.
5. Monitor cell amperage. If amperage is high, open water solenoid valve for duration to bring amperage down to the correct value.
6. Measure amp-sec on cell and integrate until desired total value is achieved.
7. Turn off cell power.
8. Open electrolyte tank drain valve for X seconds (until electrolyte tank is empty).
9. Close electrolyte tank drain valve.
10. Repeat steps 2 through 9 until the oxidant tank high level switch is active.
11. On every tenth cycle after the electrolyte tank is full, reverse polarity on the cell for five minutes, and then switch back to normal polarity and finish the cycle.

Alarm Conditions:

1. Electrolyte tank high level switch is active. Turn off power to the electrolytic cell. Turn on the red alarm LED light. Turn off brine feed solenoid valve. Turn off the water feed solenoid valve. Send alarm to display: Electrolyte tank high level switch is active.

2. Electrolyte tank high level switch is active when low level switch is not active. Turn on the red alarm LED light. Turn off brine feed solenoid. Turn off water feed solenoid. Turn off power supply. Send alarm to display: Electrolyte tank level switch failure.
3. Power supply amperage is zero when power is applied to power supply. Turn on the red LED light. Turn the power off to the power supply. Send alarm to display: Power supply amperage low.
4. Power supply amperage is too high (outside of normal electrolyte high adjustment range). Turn off power supply. Turn on the red LED alarm light. Send alarm to display: Power supply high amperage.
5. Oxidant tank Low level switch is open (oxidant is below switch level). Turn on the red LED alarm light. Turn off power supply. Activate relay to turn off external oxidant feed pumps or valves. Send alarm to display: Oxidant tank low switch is active.
6. Oxidant tank high switch is active. Turn on the red LED alarm light. Turn off power supply. Turn off the electrolyte tank valve. Send alarm to display: Oxidant high level switch is active.
7. Oxidant tank high level switch is active and oxidant tank low level switch is open. Turn on the red LED alarm light. Turn off power supply. Turn off brine and water feed solenoids. Send alarm to display: Oxidant tank level switch failure.

2015 Requirements Implementation in 2020

1. Check OT0 and OT1 signals by checking (D10, D11) ports on the Nano. With state OT0 = 1 and OT1 = 0, the system must turn on. Start a run time clock in the software. This state describes the system as having a low Oxidant level in the oxidant tank. To turn the system on, set the demux state to DCBA = 0100. DCBA represents the 4 bits in the truth table shown below. Demux states are set by sending a 1 to the ports (D2, D3, D4, D5) on the Nano. We declared state 15, or DCBA = 1111 as the 'off' state. We assume that Oxidant Tank is the final storage tank for produced oxidant, and the electrolyte tank is the median tank wherein oxidant is produced.

DCBA	State	Feature
0000	0	SV01
0001	1	SV02
0010	2	SV03
0011	3	SV04
0100	4	CELLON
0101	5	CELLP
0110	6	ALARM
0111	7	GRNLED
1000	8	YELLED
1001	9	REDLED
1010	10	-
1011	11	-
1100	12	-

1101	13	-
1110	14	-
1111	15	OFF

2. Send demux state DCBA = 0000. This will enable the SV01 Brine Tank valve. Continue in this state until the (D7) port reads 1, which means the brine tank is full. Turn off the brine tank by sending 0 to SV01, this is done by setting the demux state at DCBA = 1111.
3. Send demux state DCBA = 0001. This will enable SV02, which is the water valve. Stay in this state until OX0 = 1 and OX1 = 1. This means the Electrolyte tank is full.
4. Send demux state DCBA = 0100. This will enable CELLON and turn on the cell.
5. Read the status of port A1 on the Nano. This is current. If amperage is too high (put 20A for now), open the water valve by setting Demux state to DCBA = 0001 to release the water valve.
6. Current = I [A] = I [C/s], where Charge = Q [C]. Amps continuous signal multiply this by seconds run time. Use the run time clock started at the initialization of the system. Use the result of the multiplication, integrate over the time until desired value is achieved. There will or should be a list of pre-determined values of charge required for a certain volume of oxidant produced.
7. Set Demux state to DCBA = 1111 to turn off the cell. CELLON should read 0.
8. Send Demux state to DCBA = 0010 to activate SV03 = 1. This enables the Oxidant Drain. Maintain for X seconds, where X is the time until we get a status update from OX0 and OX1, which are ports (D8, D9) on the Nano. The state of these ports must be OX0 = 0 and OX1 = 0. Then the electrolyte tank is empty. Then set Demux state to 15, with DCBA = 1111 to turn everything off, closing the valve in the process.
9. Set Demux state to DCBA = 1111 to close all valves.
10. Every requirement exists in a ‘for’ loop, to repeat indefinitely until we get the status update OT0 = 1 and OT1 = 1 from ports (D10, D11) on the Nano.
11. After 10 iterations of the for loop, switch to reverse polarity by setting Demux state to DCBA = 0101 and keep the cell on, by sending Demux state to DCBA = 0100.

Alarm Conditions Implementation:

1. We created 6 alarm states:
    a. Alarm (0): Start-up error
    b. Alarm (1): Full error
    c. Alarm (2): Empty error
    d. Alarm (3): Sensor error, sensors being OT, OX, and BG (0,1) respectively
    e. Alarm (4): Power Supply close to zero error
    f. Alarm (5): Power Supply amps too high error

    For alarm condition #1, the system is sent to an alarm state: Alarm (1). Namely, XHH is one, if it is state OT0 = 1, OT1 = 1, then send demux state 15 to disable the cell, turn off yellow light. Turn on the red LED alarm light. Turn off brine valve (SV01). Turn off the water valve (SV02). Do all this by sending demux state 15. Display string “Electrolyte tank is full ".

Optional Diagnostic strings (not implemented, as they would most likely be within the user manual)

"Check water feed valve is closed"
"Check brine feed solenoid is closed"
"Check that switches work properly"
"Check that electrolyte tank drain operates properly"

2. Alarm (3). XHH is on and XLL is not on. Then this is DLSHH, DLSH, DLSL, DLSLL: 1110, or 1001, or 0101, or 1011, or 0110. In terms of the Nano, (OT0 = 0, OT1 = 1), or (OX0 = 0, OX1 = 1), or (BG0 = 0, BG1 = 1) are error states. Turn on Red alarm light. Set to state 15 to turn everything off.
   Send string to display: "Sensor error, check sensors"
   Optional Diagnostic strings (not implemented, as they would most likely be within the user manual)
   "Check electrolyte tank low switch for open condition when electrolyte is above low-level switch position."
   "Check oxidant tank high level switch for closed position when fluid level is below switch."
3. Alarm (4). Power supply amperage is 0 when power applied. Read CIV at port A1, pin 20 on the Nano. If the value is very close or = to 0, turn on the Red alarm light.
   -Display string: "Power supply amperage low"
   Diagnostics: "Check for salt in brine tank"
   "Check water feed to brine tank"
   "Check brine tank float value is operating properly"
   "Check power supply to system"
   "Check power at power supply"
   "Check power supply output"
   "Check brine feed solenoid operates properly"
   "Check water feed solenoid valve to electrolyte tank operates properly"
   "Check salinity of electrolyte"
4. Alarm (5). Power supply Amps are too high.
   If CIV at Nano pin 20 is too high (some value we don't know yet), then set Demux state to 15 to shut everything off.
   -Turn on red alarm light
   Display: "Power supply high amperage"
   Diagnostics: "Check brine feed solenoid is operating properly"
   "Check cell for short circuit condition"
5. Alarm (2). Oxidant tank LL level switch open. This is state (OT0 = 0, OT1 = 0)
   -Turn on Red LED
   -Turn off everything w/ Demux state 15.

Display: "oxidant tank low switch active"
Diagnostics: "Check oxidant tank for leaks"
"Ensure oxidant tank feed rate from oxidant tank is at a flow rate slower than oxidant makeup rate"
"Check salt level in brine tank"

“Check electrolyte makeup system”
“Check that electrolyte tank drain valve operates properly”

6. Alarm (1). OT HH active. This is DLSHH, DLSH, DLSL, DLSLL = 1111. This is state (OT0 = 1, OT1 = 1), or pins D10 & D11 on the Nano. Turn on Red LED, set demux to state 15.
   Display: “Oxidant tank High switch active.”
   Diagnostics: “Check for back flow of water or oxidant to oxidant tank”
   “Check electrolyte drain tank is okay”
   “Check oxidant tank high level switch”
   “Check oxidant tank high level switch operates properly and signal gets to controller
7. Alarm (3). Oxidant Tank HH switch active, while oxidant tank LL switch is open. This is a sensor error, being one of these states: (OT0 = 0, OT1 = 1), or (OX0 = 0, OX1 = 1), or (BG0 = 0, BG1 = 1). Turn on the Red LED. Send demux state to 15 to turn everything off.
   Display: “Oxidant tank level switch failure”
   Diagnostics: “Check oxidant tank low level switch for open condition when electrolyte is above Low-Level switch position”
   “Check Oxidant tank high level switch for closed position when fluid level is below switch.”

Software Implementation

Software Design Architecture

The K1 Final architecture is designed in Arduino code. It is primarily based off of a state machine and requirement decomposition.

Software Function Descriptions

Void CorrectA0(): Designed to correct the inconsistency of the analog buttons the user presses. It frequently read a different state based on what was powering the board.

Void update MuxA0(): updates the analog mux A0 which has the user buttons and sets it to the global variable ReadA0.

Void LCDdis(String a, String b, String c, String d): This function allows an easy way to program to the 4 lines of the LCD screen where string a prints to the first line, string b prints to the second, etc.

Void Hello_Phase(): this function creates a display menu for the user to select a language which then calls LanguageChange() which pulls from the FRAMs and loaded into global functions for the LCD screen to display.

Void LanguageChange(int a, int b): This function pulls the language from the fram and loads it to global variables. Int a selects which FRAM to pull from and int b is the starting address for the language.

Void Read_AB(): waits until A0 receives a single from the user. Corrects anomalies that register as below 10.

Void Actuators(int var): This function controls the digital multiplexer. Given an int var it turns on the appropriate pins for a selection.

Int Requirement1(): starts requirement 1. Looks for error states and turns on actuator 4. Returns 0 for successful completion and 1 for error. Calls alarm if error.

Int Requirement 2(): turns on Actuator 0 and waits for the tank to fill. If error signals alarm and reports error 1 to exit.

Int Requirement3(): turns on Actuator 1. And checks for error states while filling the OX tank. Turns on actuator 15 to turn everything off.

Void Requirement4(): turns on Actuator 4. Delays for 5 seconds. This is a guess.

Int Requirement56(): reads amp seconds and given if it is under 100 (also a guess) it will not do anything. If the read input amperage is a certain value, it will trigger the alarm.

Void Requirement7(): turns off everything by activating actuator 15.

Int Requirement89(): turns on actuator 2 and looks for error states. It then fills the oxidant tanks and. Turn on the alarm if error and exit. At the end it turns off everything by activating Actuator 15.

Void Requirement11(): checks the polarity count by reading from the FRAM. If it is ten it will activate actuator 10 to switch polarity and resets the count in the FRAM.

Void Requirement10(): When Begin disinfection is called this function is called. It will loop requirements 1-9 and Requirement 11 until the OT tank is full. It will first check for errors and then loop the requirements.
Int readD(int digital): will read the given digital pin and return it. Used for shorthand.

Void ampsecliters(): reads the amp seconds and returns liters. Not implemented in the code currently.

Void alarm(int a): will first turn on alarm mux and then red light. Given the error a switch statement occurs for each appropriate error. It will record it in the FRAM and send SMS. Send SMS is commented out. Includes a state where all errors can be erased.

Void errorMSG(): allows the user to select from a menu of alarms. Tell them the alarm and message of what is wrong. Also allows users to send errors to Aqua research or delete all errors.

Void LCDdis0(): Displays alarm 0 error message.
Void LCDdis1(): Displays alarm 1 error message.
Void LCDdis2(): Displays alarm 2 error message.
Void LCDdis3(): Displays alarm 3 error message.
Void LCDdis4(): Displays alarm 4 error message.
Void LCDdis5(): Displays alarm 5 error message.

Void SendSMSsetup(): is the start up for the SIM module. Commented out.

Void sendSMS(): Sends a text message to aqua research. Commented out.

Software Description

On boot, the software will run Arduinos void setup(): this will configure the mux pins, digital pins, FRAMs, and LCD screen. It will then enter the main loop. Here it will display the main screen in which 4 states are available:

Language Select, Begin disinfection, Error Message, Quit.

When language Select is selected the code will enter the language selection menu in which the user can choose from a list of languages. When a language is selected it will be pulled from the FRAM's memory and set to a global function. The user can select the menu button to go back to the main menu at any time.

When the user selects Begin Disinfection, the Arduino will begin calling the requirement functions for the disinfection protocol. It will enter requirement 10 and then go through requirements 1-9 and then requirement 11. If there is an error detected it will immediately back out of the requirement function calls.

When the user selects Error Message, the display will enter the error message menu. This will display a list of alarms and how many they have been called. If the user selects it will display an explanation for the alarm and some guidance on how to fix it. The last two options of the Error message menu are to send all errors to aqua research and clear all errors.

The last option on the main menu is Quit. This is a call yet to be implemented but is intended to turn off the system.

Software Final Code

```
/*************************************************************************************/
/*
/*  Authors:        Diego Chavez, David Kirby, John Quinlan
/*
/*  Description:    K1 disinfectant tank communication system
/*
/*  History:
/*                  v1.0 - 2015 Senior Design Group
/*                          Created LED displays for error detection
/*                  v2.0 - 2020 Senior Design Group
/*                          Added FRAM storage, LCD, and alert notification system
/*
/*************************************************************************************/

//included library section
#include <Wire.h>
#include <LiquidCrystal_PCF8574.h>
#include <SoftwareSerial.h>
#include "I2C_string.h" //Local library for strings in FRAM
//#include <SIM800.h>

//Initialization of FRAMs as string reads
I2C_string fram = I2C_string();
I2C_string fram1 = I2C_string();

//LCD display with address 0x27
LiquidCrystal_PCF8574 lcd(0x27);

//For requirements 5 and 6
int ampsec = 0;
int liters = 0;

//Analog button values
const int LL_AB = 678;
const int LM_AB = 611;
const int RM_AB = 509;
const int RR_AB = 339;

//Analog inputs
int SCLA5 = 24;
int SDAA4 = 23;
int CV4 = 22;
int TEMP1 = 21;
int CIV = 20;
int A0_Mux = A0;
int ReadA0 = 0;

//Demuxing digital pins
const int D0A = 5;
const int D1B = 6;
const int D2C = 7;
const int D3D = 8;

//digital pins definiton
const int DBG0 = 9;
const int DBG1 = 10;
const int D0X0 = 11;
const int D0X1 = 12;
const int D0T0 = 13;
const int D0T1 = 14;

//Global language. Set default as English, can be changed through language select.
String Select = "Language Select";
String BeginDisinfection = "Begin Disinfection";
String ErrorMessage = "Error Message";
String QUIT = "Quit";
```

```
//Count the number of times the cell has been used.
//Will reverse polarity every 10 times in requirement 11.
uint8_t PolarityTest;

void setup(void)
{
  //SMS set up, uncomment when fixed
  //SendSMSsetup();
  //Start FRAM i2c set up
  //Baud rate
  Serial.begin(9600);
  while (!Serial)
    ;
  {
    Serial.print("Waiting for Serial\n");
  }
  // Analog pin set up
  pinMode(A0_Mux, INPUT);
  pinMode(SCLA5, INPUT);
  pinMode(SDAA4, INPUT);
  pinMode(CV4, INPUT);
  pinMode(TEMP1, INPUT);
  pinMode(CIV, INPUT);

  //Pin Outs and Ins ------------------------------------------------------------------
  //The Mux pins
  pinMode(D0A, OUTPUT);
  pinMode(D1B, OUTPUT);
  pinMode(D2C, OUTPUT);
  pinMode(D3D, OUTPUT);

  //Other digital pins
  pinMode(DBG0, INPUT);
  pinMode(DBG1, INPUT);
  pinMode(D0X0, INPUT);
  pinMode(D0X1, INPUT);
  pinMode(D0T0, INPUT);
  pinMode(D0T1, INPUT);

  //Check the first FRAM
  if (fram1.begin(0x51))
  {
    Serial.println("Found I2C FRAM 0x51");
  }
  else
  {
    Serial.println("I2C FRAM not identified 0x51 ... check your connections?\r\n");
    alarm(0);
  }
  //Check the second FRAM
  if (fram.begin(0x50))
  { // you can stick the new i2c addr in here, e.g. begin(0x51);
    Serial.println("Found I2C FRAM 0x50");
  }
  else
  {
    Serial.println("I2C FRAM not identified 0x50... check your connections?\r\n");
    alarm(0);
  }
  //LCD set up -----------------------------------------------------------------------
  int error;
  Serial.println("LCD...\n");
  Serial.println("check for LCD");
  Wire.begin();
  Wire.beginTransmission(0x27);
  error = Wire.endTransmission();
  if (error == 0)
  {
    uint8_t test = fram.read8(0x0);
    Serial.println(": LCD found \n");
    lcd.begin(16, 2); // initialize the lcd
  }
  else
  {
    Serial.println(": LCD not found.");
    alarm(0);
  }
  lcd.setBacklight(255);
  lcd.home();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Hello LCD");
  lcd.setCursor(0, 1);
  delay(1000);
  lcd.clear();
  lcd.setBacklight(0);
}

void correctA0() //Corrects inconsistencies reading analog button
{
  //int LL_AB=678;
  //int LM_AB=611;
  //int RM_AB=509;
  //int RR_AB=339;
  if (ReadA0 < 686 && ReadA0 > 669)
  {
    ReadA0 = LL_AB;
  }
  else if (ReadA0 < 616 && ReadA0 > 609)
  {
    ReadA0 = LM_AB;
  }
  else if (ReadA0 < 511 && ReadA0 > 504)
  {
    ReadA0 = RM_AB;
  }
  else if (ReadA0 < 346 && ReadA0 > 330)
  {
    ReadA0 = RR_AB;
  }
  else
  {
    ReadA0 = 0;
  }
}
//Main loop -----------------------------------------------------------------------
void loop(void)
{
  ReadA0 = 0; //set the read A0 to zero.
  delay(1000);
  //default state selected
  int SelectOption = 1;
  LCDdis("->" + Select, BeginDisinfection, ErrorMessage, QUIT);

  //wait until button pushed
  while (ReadA0 != RM_AB)
  {
    ReadA0 = 0;
    while (ReadA0 < 10)
    {
      updateMuxA0();
      correctA0();
      //Serial.println(ReadA0);
      //delay(10);
    }
    //based on selected state go to function
    if (ReadA0 == LL_AB)
    {
      SelectOption -= 1;
    }
    if (ReadA0 == LM_AB)
    {
      SelectOption += 1;
    }
    if (SelectOption < 0)
    {
      SelectOption = 4;
    }
    if (SelectOption > 4)
    {
      SelectOption = 1;
    }

    //Selects function based on state
    switch (SelectOption)
    {
    case 1:
      LCDdis("->" + Select, BeginDisinfection, ErrorMessage, QUIT);
      break;
    case 2:
      LCDdis(Select, "->" + BeginDisinfection, ErrorMessage, QUIT);
      break;
    case 3:
      LCDdis(Select, BeginDisinfection, "->" + ErrorMessage, QUIT);
      break;
    case 4:
      LCDdis(Select, BeginDisinfection, ErrorMessage, "->" + QUIT);
      break;
    default:
      //Should not be able to get here
      Serial.println("error");
      break;
    }
    //Serial.println(ReadA0);
  }
  //Goes to the selected function
  switch (SelectOption)
  {
  case 1:
    Hello_Phase();
    break;
  case 2:
    Requirement10();
    break;
  case 3:
    errorMSG();
    break;
  case 4:
    //this is quit. needs a function to begin
    LCDdis("Language Select", "Begin Disinfection", "Error Messages", "->Quit");
    break;
  default:
    Serial.println("error");
    break;
  }
}

//Updates ReadA0 based on what user presses
void updateMuxA0()
{
  ReadA0 = analogRead(A0_Mux);
}

//Takes four strings and displays them.
void LCDdis(String a, String b, String c, String d)
{
  lcd.setBacklight(100);
  lcd.home();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print(a);
  lcd.setCursor(0, 1);
  lcd.print(b);
  lcd.setCursor(0, 2);
  lcd.print(c);
  lcd.setCursor(0, 3);
  lcd.print(d);
  delay(100);
}

//Language select
void Hello_Phase(void)
{
  ReadA0 = 0;
  delay(1000);
  int Selectt = 1;
  LCDdis(Select, "", "<-   English   ->", "");

  //While menu has not been selected
  while (ReadA0 != RM_AB)
  {
    Read_AB();
    correctA0();
    if (ReadA0 == RR_AB)
    {
      return;
    }

    if (ReadA0 == LL_AB)
    {
      Selectt -= 1;
    }
    if (ReadA0 == LM_AB)
    {
      Selectt += 1;
    }
    if (Selectt < 0)
    {
      Selectt = 4;
    }
    if (Selectt > 4)
    {
      Selectt = 1;
    }

    //Displays language options
    switch (Selectt)
    {
    case 1:
      LCDdis(Select, "", "<-   English   ->", "");
      break;
    case 2:
      LCDdis(Select, "", "<-    Spanish    ->", "");
      break;
    case 3:
      LCDdis(Select, "", "<-    French    ->", "");
      break;
    case 4:
      LCDdis(Select, "", "<-   Dutch    ->", "");
      break;
    default:
      Serial.println("error");
      break;
    }
  }
  //Change the global strings based on language
  switch (Selectt)
  {
  case 1:
    //English
    LanguageChange(0, 0);
    break;
  case 2:
    //Spanish
    LanguageChange(0, 80);
    break;
  case 3:
    //French
    LanguageChange(1, 0);
    break;
  case 4:
    //Dutch
    LanguageChange(1, 80);
    break;
  default:
    Serial.println("error");
    break;
  }
  return;
}

//Changes how to read the FRAM. Will need to change if you change languages in the FRAM
void LanguageChange(int a, int b)
{
  if (a == 0)
  {
    Select = fram.read_String(b + 0, 19);
    BeginDisinfection = fram.read_String(b + 20, 19);
    ErrorMessage = fram.read_String(b + 40, 19);
    QUIT = fram.read_String(b + 60, 19);
  }

  if (a == 1)
  {
    Select = fram1.read_String(b + 0, 19);
    BeginDisinfection = fram1.read_String(b + 20, 19);
    ErrorMessage = fram1.read_String(b + 40, 19);
    QUIT = fram1.read_String(b + 60, 19);
  }
}

//Waits for a button to be pressed by the user
void Read_AB(void)
{
  ReadA0 = 0;
  while (ReadA0 < 10)
  {
    updateMuxA0();
    //Serial.println(ReadA0);
  }
}

//Demuxing states. Will take argument int and output configuration in demux binary
void Actuators(int var)
{
  switch (var)
  {
  case 0: //Solenoid Valve 1
    digitalWrite(D0A, LOW);
    digitalWrite(D1B, LOW);
    digitalWrite(D2C, LOW);
    digitalWrite(D3D, LOW);
    break;

  case 1: //Solenoid Valve 2
```

```
    digitalWrite(D0A, HIGH);
    digitalWrite(D1B, LOW);
    digitalWrite(D2C, LOW);
    digitalWrite(D3D, LOW);
    break;

  case 2: //Solenoid Valve 3
    digitalWrite(D0A, LOW);
    digitalWrite(D1B, HIGH);
    digitalWrite(D2C, LOW);
    digitalWrite(D3D, LOW);
    break;

  case 3: //Solenoid Valve 4
    digitalWrite(D0A, HIGH);
    digitalWrite(D1B, HIGH);
    digitalWrite(D2C, LOW);
    digitalWrite(D3D, LOW);
    break;

  case 4: //CELL ON
    digitalWrite(D0A, LOW);
    digitalWrite(D1B, LOW);
    digitalWrite(D2C, HIGH);
    digitalWrite(D3D, LOW);
    break;

  case 5: //Cell Polarity
    digitalWrite(D0A, HIGH);
    digitalWrite(D1B, LOW);
    digitalWrite(D2C, HIGH);
    digitalWrite(D3D, LOW);
    break;

  case 6: //Alarm
    digitalWrite(D0A, LOW);
    digitalWrite(D1B, HIGH);
    digitalWrite(D2C, HIGH);
    digitalWrite(D3D, LOW);
    break;

  case 7: //Green LED
    digitalWrite(D0A, HIGH);
    digitalWrite(D1B, HIGH);
    digitalWrite(D2C, HIGH);
    digitalWrite(D3D, LOW);
    break;

  case 8: //Yello LED
    digitalWrite(D0A, LOW);
    digitalWrite(D1B, LOW);
    digitalWrite(D2C, LOW);
    digitalWrite(D3D, HIGH);
    break;

  case 9: //red LED
    digitalWrite(D0A, HIGH);
    digitalWrite(D1B, LOW);
    digitalWrite(D2C, LOW);
    digitalWrite(D3D, HIGH);
    break;

  case 10: //empty do nothing
    digitalWrite(D0A, LOW);
    digitalWrite(D1B, HIGH);
    digitalWrite(D2C, LOW);
    digitalWrite(D3D, HIGH);
    break;
  case 11: //empty do nothing
    digitalWrite(D0A, HIGH);
    digitalWrite(D1B, HIGH);
    digitalWrite(D2C, LOW);
    digitalWrite(D3D, HIGH);
    break;

  case 15: //default off state
    digitalWrite(D0A, HIGH);
    digitalWrite(D1B, HIGH);
    digitalWrite(D2C, LOW);
    digitalWrite(D3D, HIGH);
    break;

  default:
    Serial.print("Error. Incorrect demux state.\n");
    delay(1);
    break;
  }
}

//Code based off the 2015 K1 requirements, but "interpreted" to fit the new board
//Requirement 1
int Requirement1(void)
{
  LCDdis("Requirement 1", "started", "", "");
  delay(300);

  //Full state no error but alarm
  if (readD(D0T0) == 1 && readD(D0T1) == 1)
  {
    alarm(1);
    return 1;
  }

  //Error state defaults
  if (readD(D0T0) == 0 && readD(D0T1) == 1 || readD(DBG0) == 0 &&
      readD(DBG1) == 1 || readD(D0X0) == 0 && readD(D0X1) == 1)
  {
    alarm(3);
    return 1;
  }
  //Empty state = tell user to check tanks
  if (readD(DBG0) == 0 && readD(DBG1) == 0 || readD(D0X0) == 0 && readD(D0X1) == 0)
  {
    alarm(2);
    return 1;
  }

  //Start state = no error
  if (readD(D0T1) == 0 && readD(D0T0) == 1 || readD(D0T1) == 0 && readD(D0T0) == 0)
  {
    Serial.print("start state in req1\n"); //Turn system on
    delay(5000);
    Actuators(4); //Turn on cell
    return 0;     //Return with no error
  }
}

int Requirement2(void)
{
  LCDdis("Requirement 2", "started", "", "");
  Actuators(0);             //Turn on solenoid valve 1
  while (readD(DBG1) == 1) //while empty wait to fill
  {
    //do nothing}
    delay(160);
    Serial.print("in Req2 waiting for BG1\n");
  }
  //error state
  if (readD(DBG1) == 1 && readD(DBG0) == 0)
  {
    alarm(3);
    return 1;
  }
  Serial.print("Req2 success\n");
  //no error reported. turn off everything
  Actuators(15);
  return 0;
}

int Requirement3(void)
{
  LCDdis("Requirement 3", "started", "", "");
  delay(1600);
  Actuators(1);                                //Turn on solenoid valve 1
  while (readD(D0X0) != 1 && readD(D0X1) != 1) //While not full
  {

    if (readD(D0X0) == 0 && readD(D0X1) == 1) //error state in sensors
    {
      alarm(3);
      return 1;
    }
    delay(1);
  }
  Actuators(15);
  return 0;
}

void Requirement4(void)
{
  LCDdis("Requirement 4", "started", "", "");
  //Cell on
  Actuators(4);
  delay(5000);
  //Actuators(15);
}

//Read amps-sec and output as liters. Not defined yet by Tim
int Requirement56(void)
{
  LCDdis("Requirement 56", "started", "", "");
  int Temp = 0;
  int sec = 0;

  //While amp-sec is less than a certain value
  while (ampsec < 100)
  {
    delay(1000);
    Temp = analogRead(CIV);
    //Error if CIV amperage is very low
    if (analogRead(Temp <= 1))
    {
      alarm(4);
      return 1;
    }
    //Error if CIV is way too high
    if (analogRead(Temp >= 5))
    {
      alarm(5);
      return 1;
    }

    ampsec = Temp * sec;
    //100? 100 is supposed to be charge
    sec += 1;
  }
  return 0;
}

//Turn off everything
void Requirement7(void)
{
  LCDdis("Requirement 10", "started", "", "");
  Actuators(15);
}

int Requirement89(void)
{
  LCDdis("Requirement 89", "started", "", "");
  Actuators(2);                                //Turn on solenoid valve 3
  while (readD(D0X0) != 0 && readD(D0X1) != 0) //While not empty
  {
    //Checking errors in sensors
    if (readD(D0T0) == 0 && readD(D0T1) == 1 || readD(DBG0) == 0 &&
        readD(DBG1) == 1 || readD(D0X0) == 0 && readD(D0X1) == 1)
    {
      alarm(3);
      return 1;
    }
    delay(1000);
  }
  //Turn off everything and report no error
  Actuators(15);
  return 0;
}
//Change polarity every ten uses. Saves to fram and counts between shut offs
void Requirement11(void)
{
  LCDdis("Requirement 11", "started", "", "");
  //Read from address in FRAM
  PolarityTest = fram.read8(0xFF);
  if (PolarityTest > 9) //If greater then 9, reset
  {
    Actuators(10);
    PolarityTest = 0;
    delay(1000);
    Actuators(15);
  }
  PolarityTest += 1;
  fram.write8(0xFF, PolarityTest); //Save state to FRAM
}
//Generating
void Requirement10(void)
{
  LCDdis("Requirement 10", "started", "", "");
  delay(3000);
  //Tank full
  if (readD(D0T1) == 1 && readD(D0T0) == 1)
  {
    LCDdis("Oxidant Tank Full", "Empty Please", "", "");
    return;
  }
  //Error states for sensors
  if (readD(D0T0) == 0 && readD(D0T1) == 1 || readD(DBG0) == 0 &&
      readD(DBG1) == 1 || readD(D0X0) == 0 && readD(D0X1) == 1)
  {
    alarm(3);
    return;
  }
  //Error states for empty tanks
  if (readD(DBG0) == 0 && readD(DBG1) == 0 || readD(D0X0) == 0 && readD(D0X1) == 0)
  {
    alarm(2);
    return;
  }
  //While OT tank is not full, generate
  while (readD(D0T1) != 1 && readD(D0T0) != 1)
  {
    //If a state returns 1, it will exit this function
    if (Requirement1() == 1)
    {
      return;
    }
    if (Requirement2() == 1)
    {
      return;
    }
    if (Requirement3() == 1)
    {
      return;
    }
    Requirement4();
    if (Requirement56() == 1)
    {
      return;
    }
    Requirement7();
    if (Requirement89() == 1)
    {
      return;
    }
    Requirement11();
  }
}
//Reads a digital pin
int readD(int digital)
{
  return digitalRead(digital);
}

//Converts the ampseconds to liters for output
void ampsecliters(void)
{
  if (ampsec > 2)
  {
    liters = 20;
  }
  if (ampsec > 1.5)
  {
    liters = 15;
  }
  if (ampsec > 1)
  {
    liters = 10;
  }
  if (ampsec > .5)
  {
    liters = 5;
  }
}
//Activate alarm if conditions are met
void alarm(int a)
{
  uint8_t temp = 0;
  //Activate alarm
  Actuators(6);
  //Turn on red light
  Actuators(10);
  //Record error, tell user what is wrong, send SMS
  switch (a)
  {
  //Start up error
  case 0:
    LCDdis0();
    temp = fram1.read8(0xF8);
    temp += 1;
    fram1.write8(0xF8, temp);
    //sendSMS((char) 0xF8);
    break;

  //Full alarm
  case 1:
    LCDdis1();
    temp = fram1.read8(0xF9);
    temp += 1;
    fram1.write8(0xF9, temp);
    //sendSMS((char) 0xF9);
    break;

  //Empty error
  case 2:
    LCDdis2();
    temp = fram1.read8(0xFA);
    temp += 1;
    fram1.write8(0xFA, temp);
    //sendSMS((char) 0xFA);
    break;

  //Sensor error illegal state
  case 3:
    LCDdis3();
    temp = fram1.read8(0xFB);
    temp += 1;
    fram1.write8(0xFB, temp);
    //sendSMS((char) 0xFB);
    break;

  //Power supply close-to-zero error
  case 4:
    LCDdis4();
    temp = fram1.read8(0xFC);
    temp += 1;
    fram1.write8(0xFC, temp);
    //sendSMS((char) 0xFC);
    break;

  //Power supply over amperage
  case 5:
    LCDdis5();
    temp = fram.read8(0xFD);
    temp += 1;
    fram1.write8(0xFD, temp);
    //sendSMS((char) 0xFD);
    break;

  //Clear state for whatever reason
  default:
    fram1.write8(0xF8, 0);
    fram1.write8(0xF9, 0);
    fram1.write8(0xFA, 0);
    fram1.write8(0xFB, 0);
    fram1.write8(0xFC, 0);
    fram1.write8(0xFD, 0);
    break;
    //sendSMS((char) a);
  }
}
//User presses this from main menu.
//Tells user the recorded errors, what alarms went off, and what to do.
void errorMSG(void)
{
  //Default state
  ReadA0 = 0;
```

```
delay(1000);
int Selectt = 1;
LCDdis(ErrorMessage, "<-Alarm 0->", "Start up error", "number " +
      (String)fram1.read8(0xF8));
//While main menu is not pressed
while (ReadA0 != RM_AB)
{
  Read_AB();
  correctA0();
  if (ReadA0 == RR_AB)
  {
    return;
  }
  if (ReadA0 == LL_AB)
  {
    Selectt -= 1;
  }
  if (ReadA0 == LM_AB)
  {
    Selectt += 1;
  }
  if (Selectt < 0)
  {
    Selectt = 8;
  }
  if (Selectt > 8)
  {
    Selectt = 1;
  }

  switch (Selectt)
  {
  case 1:
    LCDdis(ErrorMessage, "<-Alarm 0->", "Start up error", "number " +
          (String)fram1.read8(0xF8));
    break;
  case 2:
    LCDdis(ErrorMessage, "<-Alarm 1->", "Full Error", "number " +
          (String)fram1.read8(0xF9));
    break;
  case 3:
    LCDdis(ErrorMessage, "<-Alarm 2->", "Tank Empty", "number " +
          (String)fram1.read8(0xFA));
    break;
  case 4:
    LCDdis(ErrorMessage, "<-Alarm 3->", "Sensor Error", "number " +
          (String)fram1.read8(0xFB));
    break;
  case 5:
    LCDdis(ErrorMessage, "<-Alarm 4->", "Power Supply Low", "number " +
          (String)fram1.read8(0xFC));
    break;
  case 6:
    LCDdis(ErrorMessage, "<-Alarm 5->", "Power Supply High", "number " +
          (String)fram1.read8(0xFD));
    break;
  case 7:
    LCDdis(ErrorMessage, "<-       ->", "Send Errors", "To Aqua Research");
    break;
  case 8:
    LCDdis(ErrorMessage, "<-       ->", "Clear all Errors", "number " +
          (String)(fram1.read8(0xF8) + fram1.read8(0xF9) + fram1.read8(0xFA) +
          fram1.read8(0xFB) + fram1.read8(0xFC) + fram1.read8(0xFD)));
    break;
  default:
    Serial.println("error");
    break;
  }
}
```

```
  switch (Selectt)
  {
  case 1:
    //Alarm 0 message
    LCDdis0();
    break;
  case 2:
    //Alarm 1 message
    LCDdis1();
    break;
  case 3:
    //Alarm 2 message
    LCDdis2();
    break;
  case 4:
    //Alarm 3 message
    LCDdis3();
    ReadA0 = 0;
    while (ReadA0 < 10)
    {
      Read_AB();
    }
    break;
  case 5:
    //Alarm 4 message
    LCDdis4();
    break;
  case 6:
    //Alarm 5 message
    LCDdis5();
    break;
  case 7:
    //sends all errors to Aqua Research
    for (int i = 0xF8; i < 0xFE; i++)
    {
      if (fram1.read8(i) != 0)
      {
        delay(2);
        //sendSMS((char)i)
      }
    }
    delay(1);
    break;
  case 8:
    //clear all errors
    alarm(15);
    break;
  default:
    Serial.println("error");
    break;
  }
  return;
}

//Alarm 0 message.
void LCDdis0()
{
  lcd.setBacklight(100);
  lcd.home();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Start up Error");
  lcd.setCursor(0, 1);
  lcd.print("");
  lcd.setCursor(0, 2);
  lcd.print("");
  lcd.setCursor(0, 3);
  lcd.print("");
  delay(100);
  ReadA0 = 0;
  while (ReadA0 < 10)
  {
    Read_AB();
  }
}
//Alarm 1 message
void LCDdis1()
{
  lcd.setBacklight(100);
  lcd.home();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Empty Full Tank");
  lcd.setCursor(0, 1);
  lcd.print("Check Tank:");
  lcd.setCursor(0, 2);
  lcd.print("Brine, Oxidant, Or");
  lcd.setCursor(0, 3);
  lcd.print("Electrolyte Tanks");
  delay(100);
  ReadA0 = 0;
  while (ReadA0 < 10)
  {
    Read_AB();
  }
}
//Alarm 2 message
void LCDdis2()
{
  lcd.setBacklight(100);
  lcd.home();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Check for Empty Tank");
  lcd.setCursor(0, 1);
  lcd.print("Check Tank");
  lcd.setCursor(0, 2);
  lcd.print("Brine and");
  lcd.setCursor(0, 3);
  lcd.print("Electrolyte");
  delay(100);
  ReadA0 = 0;
  while (ReadA0 < 10)
  {
    Read_AB();
  }
}
//Alarm 3 message
void LCDdis3()
{
  lcd.setBacklight(100);
  lcd.home();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Sensor Error");
  lcd.setCursor(0, 1);
  lcd.print("Check all Tank");
  lcd.setCursor(0, 2);
  lcd.print("Sensors");
  lcd.setCursor(0, 3);
  lcd.print("");
  delay(100);
  ReadA0 = 0;
  while (ReadA0 < 10)
  {
    Read_AB();
  }
}
```

```
//Alarm 4 message
void LCDdis4()
{
  lcd.setBacklight(100);
  lcd.home();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("No Current");
  lcd.setCursor(0, 1);
  lcd.print("From Power Supply");
  lcd.setCursor(0, 2);
  lcd.print("");
  lcd.setCursor(0, 3);
  lcd.print("");
  delay(100);
  ReadA0 = 0;
  while (ReadA0 < 10)
  {
    Read_AB();
  }
}
//Alarm 5 message
void LCDdis5()
{
  lcd.setBacklight(100);
  lcd.home();
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Excess Current");
  lcd.setCursor(0, 1);
  lcd.print("Power Supply Too");
  lcd.setCursor(0, 2);
  lcd.print("Much Amperage");
  lcd.setCursor(0, 3);
  lcd.print("");
  delay(100);
  ReadA0 = 0;
  while (ReadA0 < 10)
  {
    Read_AB();
  }
}
/* When sim card is hooked up properly uncomment
//Set up for SMS
void SendSMSsetup(void)
{
  SIM.begin(9600);
  delay(100);
  SIM.pinCode(GET);
  // WARNING! Be certain that you input the correct pin code!
  if (SIM.reply("SIM PIN")) SIM.pinCode(SET, code);
}
//Sends the SMS
void sendSMS(char error){
  SIM.smsFormat(SET, "1");
  SIM.smsSend(addr, error);
}
*/
```

Additional Headers for the Final K1 Code

I2C_String.cpp

```
/**************************************************************************/
/*!
    @file     Adafruit_FRAM_I2C.cpp
    @author   KTOWN (Adafruit Industries)
    @license  BSD (see license.txt)

    Driver for the Adafruit I2C FRAM breakout.

    Adafruit invests time and resources providing this open source code,
    please support Adafruit and open-source hardware by purchasing
    products from Adafruit!

    @section  HISTORY

    v1.0 - First release
    v1.1 - 16-bit expansion using bitwise operation
*/
/**************************************************************************/
//#include <avr/pgmspace.h>
//#include <util/delay.h>
#include <stdlib.h>
#include <math.h>

#include "I2C_string.h"

/*========================================================================*/
/*                            CONSTRUCTORS                                */
/*========================================================================*/

/**************************************************************************/
/*!
    Constructor
*/
/**************************************************************************/
I2C_string::I2C_string(void)
{
  _framInitialised = false;
}

/*========================================================================*/
/*                           PUBLIC FUNCTIONS                             */
/*========================================================================*/

/**************************************************************************/
/*!
    Initializes I2C and configures the chip (call this function before
    doing anything else)
*/
/**************************************************************************/
boolean I2C_string::begin(uint8_t addr)
{

  i2c_addr = addr;
  Wire.begin();

  /* Make sure we're actually connected */
  uint16_t manufID, prodID;
  getDeviceID(&manufID, &prodID);
  if (manufID != 0x00A)
  {
    Serial.print("Unexpected Manufacturer ID: 0x");
    Serial.println(manufID, HEX);
    return false;
  }
  if (prodID != 0x510)
  {
    Serial.print("Unexpected Product ID: 0x");
    Serial.println(prodID, HEX);
    return false;
  }

  /* Everything seems to be properly initialized and connected */
  _framInitialised = true;

  return true;
}

/**************************************************************************/
/*!
    @brief  Writes a byte at the specific FRAM address

    @params[in] i2cAddr
                The I2C address of the FRAM memory chip (1010+A2+A1+A0)
    @params[in] framAddr
                The 16-bit address to write to in FRAM memory
    @params[in] i2cAddr
                The 8-bit value to write at framAddr
*/
/**************************************************************************/
void I2C_string::write8 (uint16_t framAddr, uint8_t value)
{
  Wire.beginTransmission(i2c_addr);
  Wire.write(framAddr >> 8);
  Wire.write(framAddr & 0xFF);
  Wire.write(value);
  Wire.endTransmission();
}

/**************************************************************************/
/*!
    @brief  Reads an 8-bit value from the specified FRAM address

    @params[in] i2cAddr
                The I2C address of the FRAM memory chip (1010+A2+A1+A0)
    @params[in] framAddr
                The 16-bit address to read from in FRAM memory

    @returns    The 8-bit value retrieved at framAddr
*/
/**************************************************************************/
uint8_t I2C_string::read8 (uint16_t framAddr)
{
  Wire.beginTransmission(i2c_addr);
  Wire.write(framAddr >> 8);
  Wire.write(framAddr & 0xFF);
  Wire.endTransmission();

  Wire.requestFrom(i2c_addr, (uint8_t)1);

  return Wire.read();
}

/**************************************************************************/
/*!
    @brief  Reads the Manufacturer ID and the Product ID from the IC

    @params[out]  manufacturerID
                  The 12-bit manufacturer ID (Fujitsu = 0x00A)
    @params[out]  productID
                  The memory density (bytes 11..8) and proprietary
                  Product ID fields (bytes 7..0). Should be 0x510 for
                  the MB85RC256V.
```

```
*/
/**************************************************************************/
void I2C_string::getDeviceID(uint16_t *manufacturerID, uint16_t *productID)
{
  uint8_t a[3] = { 0, 0, 0 };
  uint8_t results;

  Wire.beginTransmission(MB85RC_SLAVE_ID >> 1);
  Wire.write(i2c_addr << 1);
  results = Wire.endTransmission(false);

  Wire.requestFrom(MB85RC_SLAVE_ID >> 1, 3);
  a[0] = Wire.read();
  a[1] = Wire.read();
  a[2] = Wire.read();

  /* Shift values to separate manuf and prod IDs */
  /* See p.10 of http://www.fujitsu.com/downloads/MICRO/fsa/pdf/products/memory/fram/MB85RC256V-
DS501-00017-3v0-E.pdf */
  *manufacturerID = (a[0] << 4) + (a[1]  >> 4);
  *productID = ((a[1] & 0x0F) << 8) + a[2];
}

/**************************************************************************/
/*!
    @brief  Writes a 16-bit value to the specified FRAM address

    @params[in] framAddr
                The I2C address of the FRAM memory chip (1010+A2+A1+A0)
    @params[in] value1
                The 16-bit value to be written

*/
/**************************************************************************/
void I2C_string::write16(uint16_t framAddr, uint16_t value1)
{
  int c_address = framAddr+framAddr;
  uint16_t low,high;
  low=value1 & 0x00FF;
  high=value1 & 0xFF00;
  high= high >> 8;

  write8(c_address, low);
  write8(c_address+1, high);
}

/**************************************************************************/
/*!
    @brief  Reads a 16-bit value from the specified FRAM address

    @params[in] framAddr
                The 8-bit address to read from in FRAM memory

    @returns    The 16-bit value retrieved at framAddr
*/
/**************************************************************************/
uint16_t I2C_string::read16(uint16_t framAddr)
{
  int c_address = framAddr+framAddr;
  uint16_t low;
  uint16_t high;
  low = read8(c_address);//& 0xff;
  high = read8(c_address+1);// << 8;

  uint16_t temp1 = 256*high+low;

  return twos_comp_check(temp1);
}
```

```
/***************************************************************************/
/*!
    @brief  Checks to see if signed value is negative or positive

    @params[in] number
                The value to be checked

    @returns    The value as either a correct negative number or a positive
*/
/***************************************************************************/
uint16_t I2C_string::twos_comp_check(uint16_t number)
{
  uint16_t numcheck = 0, number2 = number&0xFFFF;

  numcheck = number2 >>15;
  if(numcheck == 0)
  {
    return number;
  }
  else
  {
    number2 = number2^0xFFFF;
    return -(number2+1);
  }
}
/***************************************************************************/
/*!
    @brief  Reads a string from the specified FRAM address

    @params[in] Addr
                The I2C address of the FRAM memory chip (1010+A2+A1+A0)
    @params[in] length
                The length of the string being read

    @returns    The string read from starting address ending at length
*/
/***************************************************************************/
String I2C_string::read_String(int Addr, int length)
{
  char temparray[length];
  for(int i=0;i<length;i++)
  {
    temparray[i]= (char)read8(Addr+i);
  }
  String stemp(temparray);

  return stemp;
}
/***************************************************************************/
/*
/*    @brief Writes string to the specified FRAM address
/*
/*    @params[in] Addr
/*              The I2C address of the FRAM memory chip (1010+A2+A1+A0)
/*    @params[in] input
/*              The String to be written to FRAM memory
/*
/*
/***************************************************************************/
void I2C_string::write_String(int Addr, String input)
{
  int numBytes;
  char cbuff[input.length()+1];
  input.toCharArray(cbuff,input.length()+1);
  for (int i = 0; i < input.length()+1; i++) {
    write8(Addr + i,cbuff[i]);
  }

}
```

I2C_String.h

```
/**************************************************************************/
/*!
    @file     Adafruit_FRAM_I2C.h
    @author   KTOWN (Adafruit Industries)

    @section LICENSE

    Software License Agreement (BSD License)

    Copyright (c) 2013, Adafruit Industries
    All rights reserved.

    Redistribution and use in source and binary forms, with or without
    modification, are permitted provided that the following conditions are met:
    1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.
    2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.
    3. Neither the name of the copyright holders nor the
    names of its contributors may be used to endorse or promote products
    derived from this software without specific prior written permission.

    THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ''AS IS'' AND ANY
    EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
    WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
    DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY
    DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
    (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
    ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
    (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
    SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

    Revision
    1. 16-bit expansion using bitwise operation
*/
/**************************************************************************/
#ifndef _I2C_string_H_
#define _I2C_string_H_

#if ARDUINO >= 100
 #include <Arduino.h>
#else
 #include <WProgram.h>
#endif

#include <Wire.h>

#define MB85RC_DEFAULT_ADDRESS        (0x50) /* 1010 + A2 + A1 + A0 = 0x50 default */
#define MB85RC_SLAVE_ID       (0xF8)

class I2C_string {
 public:
  I2C_string(void);

  boolean  begin(uint8_t addr = MB85RC_DEFAULT_ADDRESS);
  void     write8 (uint16_t framAddr, uint8_t value);
  uint8_t  read8  (uint16_t framAddr);
  void     getDeviceID(uint16_t *manufacturerID, uint16_t *productID);

  //Added functions
  void write16(uint16_t framAddr, uint16_t);
  uint16_t read16(uint16_t framAddr);
  uint16_t twos_comp_check(uint16_t);

  String read_String(int, int);
  void write_String(int, String);

 private:
  uint8_t i2c_addr;
  boolean _framInitialised;
};

#endif
```

I2C FRAM Language Programmer

```
////Created by Diego Chavez Spring 2020 For senior design 2
//The purpose of this is to write a language too two different FRAMs
//although my reading and writing to FRAMs is not complex it is also not efficient for memory
#include "I2C_string.h"
#include <Wire.h>

void setup() {
  Serial.begin(9600);
  I2C_string fram = I2C_string();
  //fram 0
  fram.begin(0x50);
  I2C_string fram1 = I2C_string();
  //fram1
  fram1.begin(0x51);
  //+2 for null string
  String EnglishLS = "Language Select   ";
  //Spanish ls is 18 long but next needs to be written at 20
  fram.write_String(0,EnglishLS);

  String EnglishBD = "Begin Disinfection";
  fram.write_String(20,EnglishBD);

  String EnglishEM = "Error Message";
  fram.write_String(40,EnglishEM);

  String EnglishQ = "Quit";
  fram.write_String(60,EnglishQ);
  //+1 for the null character when reading. Spanish ls is 18 +null is 19 long
 Serial.println(fram.read_String(0,19));
 Serial.println(fram.read_String(20,19));
 Serial.println(fram.read_String(40,19));
 Serial.println(fram.read_String(60,19));


  String SpanishLS = "Seleccionar idioma";
  //SpanishLS = "Language Select   ";
  //Spanish ls is 18 long but next needs to be written at 20
  fram.write_String(80,SpanishLS);
  //it does not like the accent mark when displaying on lcd
  String SpanishBD = "desinfección";
  //SpanishBD = "Begin Disinfection";
  fram.write_String(100,SpanishBD);

  String SpanishEM = "Mensaje de error";
  //SpanishEM = "Error Message";
  fram.write_String(120,SpanishEM);

  String SpanishQ = "dejar";
  //SpanishEM = "Error Message";
  fram.write_String(140,SpanishQ);

  //+1 for the null character when reading. Spanish ls is 18 +null is 19 long
 Serial.println(fram.read_String(80,19));
 Serial.println(fram.read_String(100,19));
 Serial.println(fram.read_String(120,19));
 Serial.println(fram.read_String(140,19));

  //start of the second fram
  Serial.print("\n");
  delay(100);

  //+2 for null string
  String FrenchLS = "Choix de la langue";
  //Spanish ls is 18 long but next needs to be written at 20
  fram1.write_String(0,FrenchLS);

  String FrenchBD = "desinfection";
  fram1.write_String(20,FrenchBD);

  String FrenchEM = "Message d'erreur";
  fram1.write_String(40,FrenchEM);

  String FrenchQ = "quitter";
  fram1.write_String(60,FrenchQ);

  //+1 for the null character when reading. Spanish ls is 18 +null is 19 long
 Serial.println(fram1.read_String(0,19));
 Serial.println(fram1.read_String(20,19));
 Serial.println(fram1.read_String(40,19));
 Serial.println(fram1.read_String(60,19));


  String DutchLS = "Taalkeuze";
  //SpanishLS = "Language Select   ";
  //Spanish ls is 18 long but next needs to be written at 20
  fram1.write_String(80,DutchLS);

  String DutchBD = "desinfectie";
  //SpanishBD = "Begin Disinfection";
  fram1.write_String(100,DutchBD);

  String DutchEM = "foutmelding";
  //SpanishEM = "Error Message";
  fram1.write_String(120,DutchEM);

  String DutchQ = "stoppen";
  //SpanishEM = "Error Message";
  fram1.write_String(140,DutchQ);
  //+1 for the null character when reading. Spanish ls is 18 +null is 19 long
 Serial.println(fram1.read_String(80,19));
 Serial.println(fram1.read_String(100,19));
 Serial.println(fram1.read_String(120,19));
 Serial.println(fram1.read_String(140,19));

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Hardware

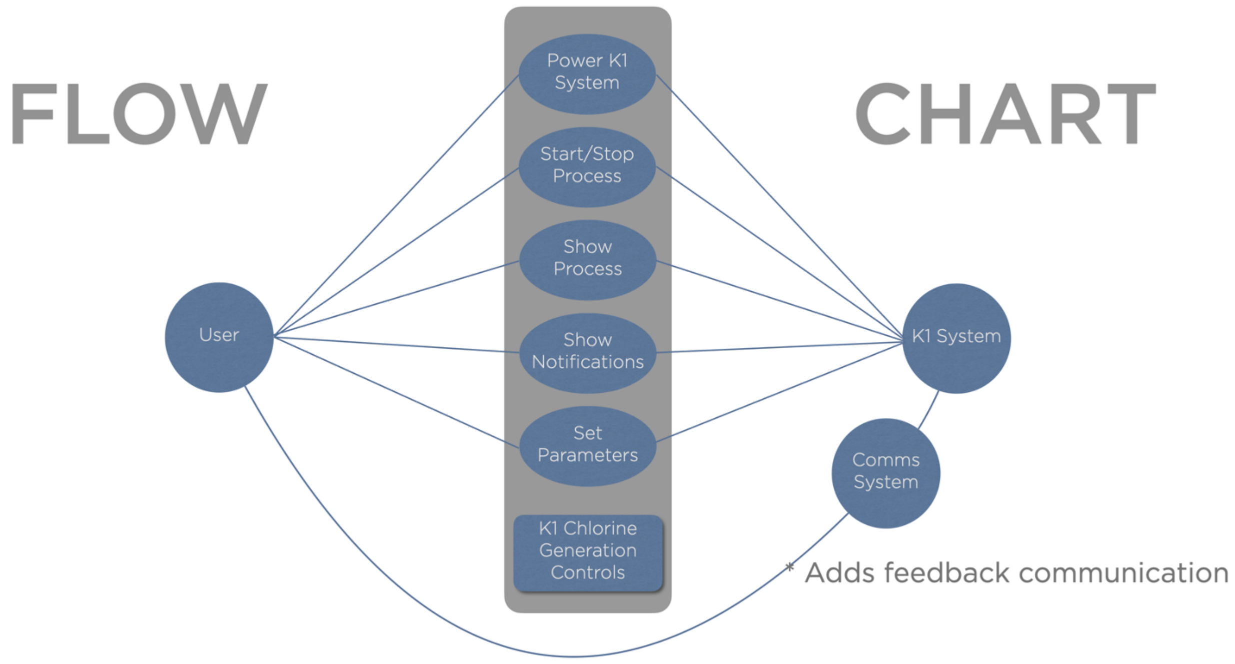

Figure 1: Intended user and K1 interactions.

Technical Documentation/ Wiring Diagrams

For the purposes of design flow, the activation ports for the solenoid valves were switched to the following combinations:

Port	SV01	SV02	SV03
Previous Declaration	Water	Brine	Electrolyte
New Declaration	Brine	Water	Electrolyte

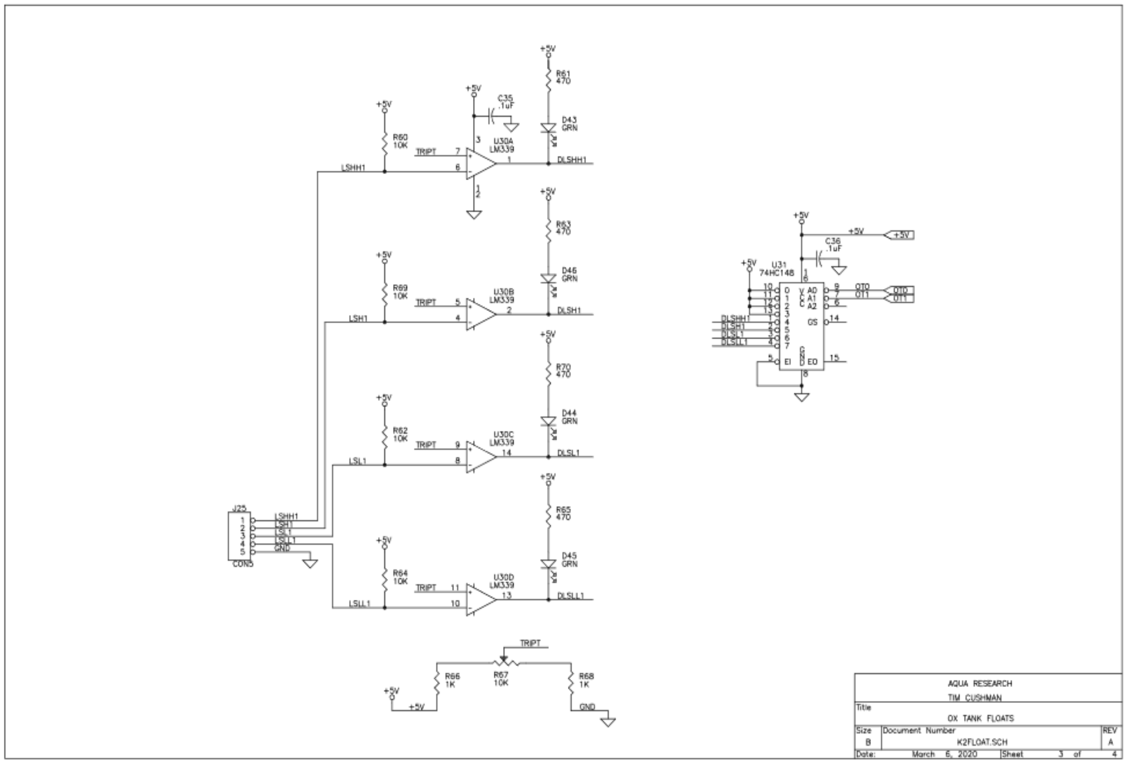

Figure 2: Ox Tank Floats

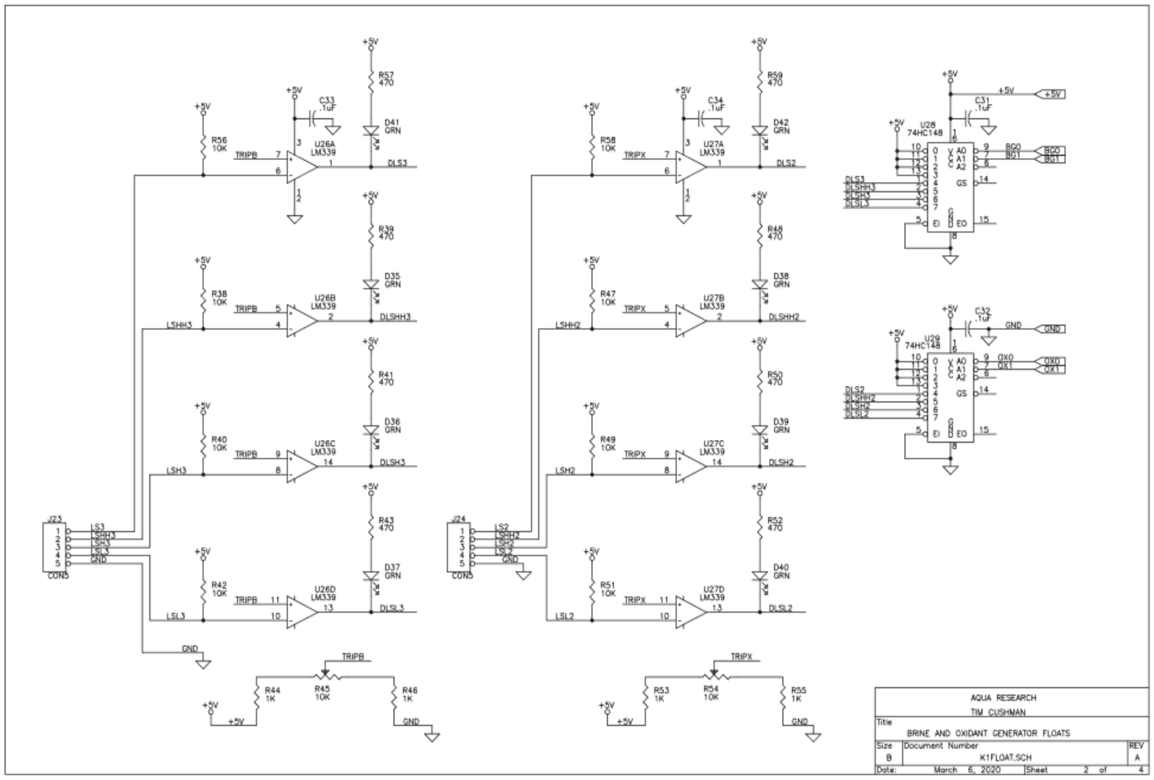

Figure 3: Brine and Oxidant Generator Floats

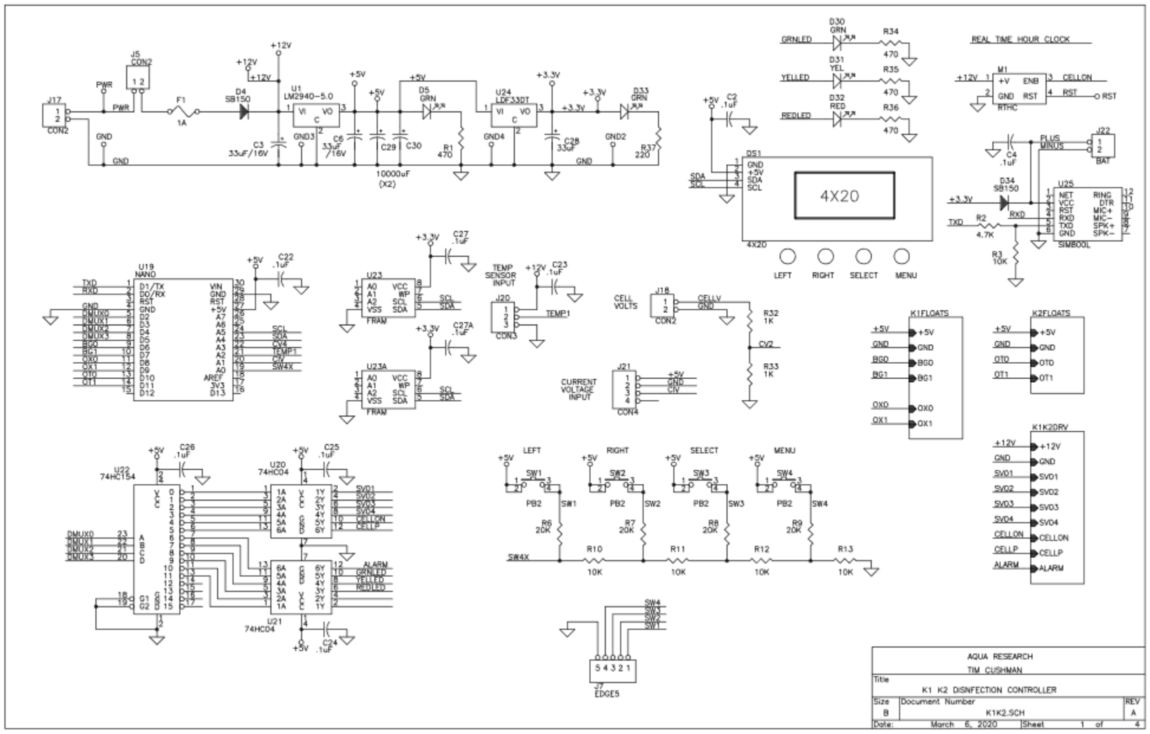
Figure 4: K1 System Schematic

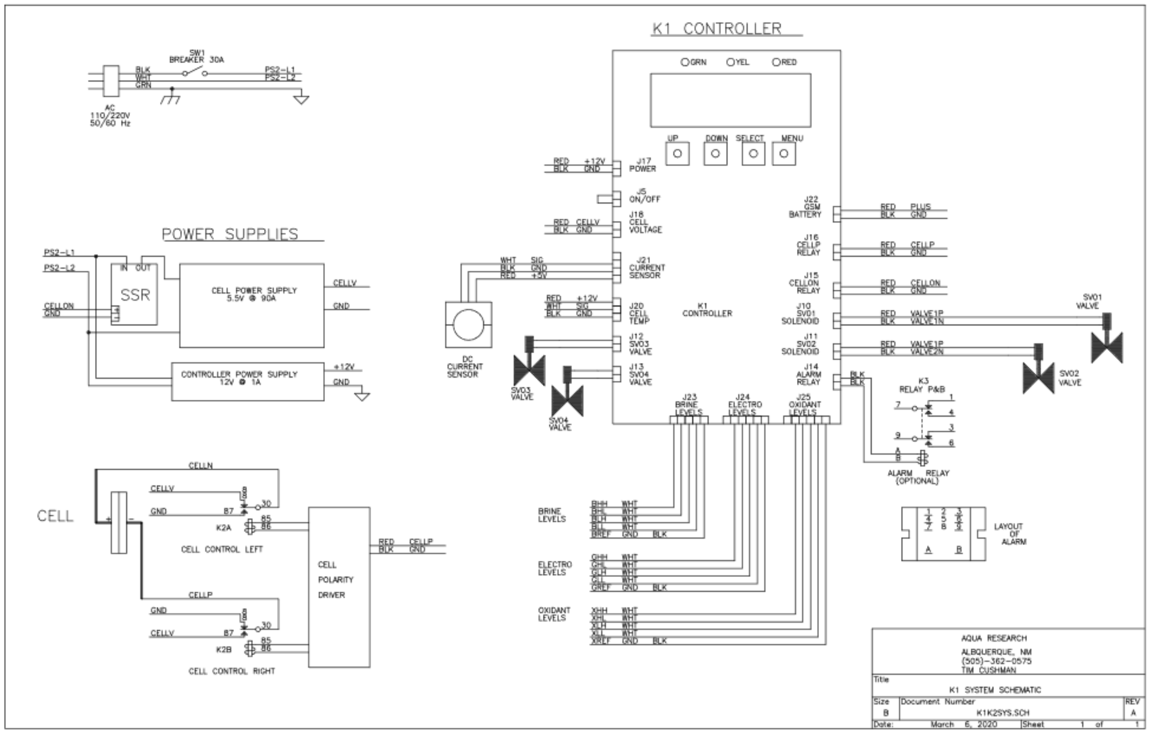
Figure 5: K1 K2 Disinfection Controller

Abbreviations

FRAM	Ferroelectric Random-Access Memory
I2C	**I2C** is a multi-master bus, which means that multiple chips can be connected to the same bus and each one can act as a master by initiating a data transfer.